RPTU
Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau

Department of
Computer Science

Kaiserslautern
2025

3D Scene Reconstruction

Group 27:
Daiana Tei
Dhruv Kale
Ahsan Aftab

Course:
3D Computer Vision (2024/25)
Module INF-73-51-M-5
Prof. Dr. Didier Stricker

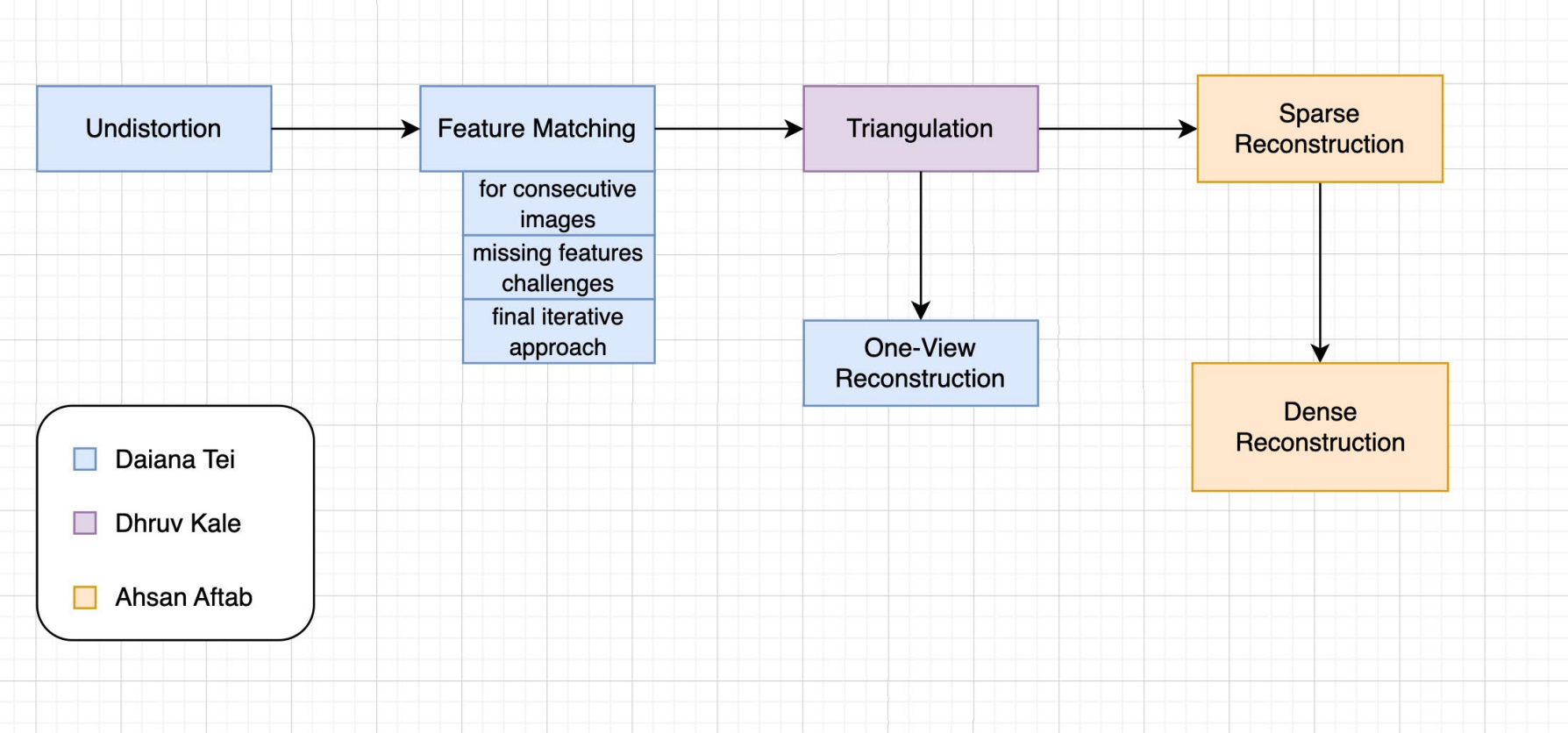
Task Overview

The goal of the exercise is to do a dense reconstruction of a scene, given colour (distorted) and depth images, camera calibration and transformation parameters, as well as a set of non-matched features.

Subtasks:

- Image undistortion
- Feature matching
- Feature points triangulation
- One-view sparse reconstruction
- Dense reconstruction

Group & Task Management

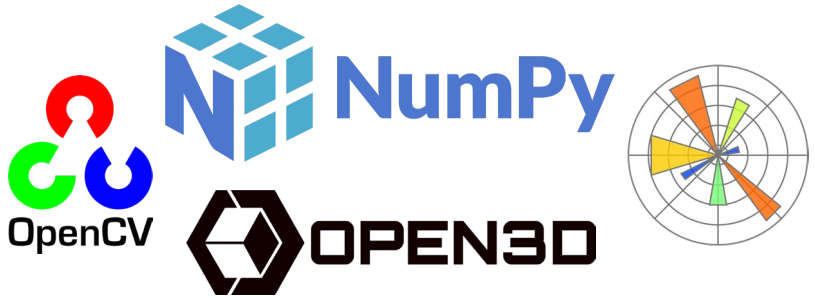


Tools & Technologies

- Implementation in Python 3.11
- Via Anaconda platform
- Using Jupyter Notebook
- Collaborating via GitHub

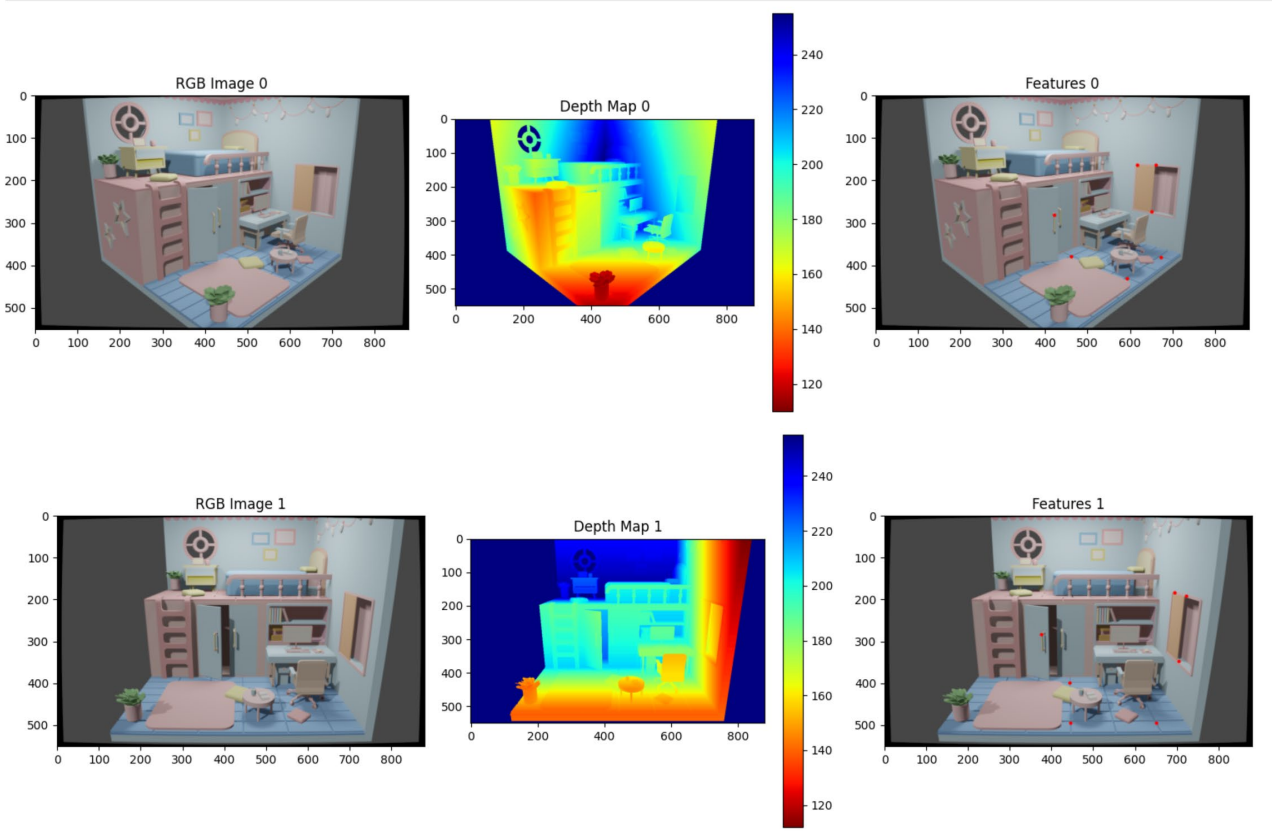
Libraries:

- NumPy for math
- OpenCV for data visualisation
- Matplotlib for intermediate plots
- Open3D for creating 3D scenes
- Os for interaction with files



Data Observation

```
Camera Calibration:
  distortion_param: [-0.1, 0.02, 0.0, 0.0, -0.01]
  image_height: 551
  image_width: 881
  principal_point: [275.0, 440.0]
  focal_length_mm: 25
  sensor_width_mm: 35
  pixel_ratio: 1.0
  pixel_per_mm: 25.17142857142857
  focal_length_px: 629.2857142857142
Camera Movement: (8, 4, 4)
2D Features (Unsorted): (9, 7, 2)
```



- 9 images with lens distortion
- 9 corresponding undistorted non-connected depth images
- Camera calibration parameters
- Distortion parameters
- Matrices of camera transformations between consecutive images
- Arrays of 7 features detected in each image

Undistortion

Brown’s Distortion Model:

$$
\begin{aligned}
x' &= x + \underbrace{x(k_1 r^2 + k_2 r^4 + k_3 r^6)}_{\Delta x_{radial}} + \underbrace{2p_1 xy + p_2(r^2 + 2x^2)}_{\Delta x_{tangential}} \\
&= x + \Delta x_{radial} + \Delta x_{tangential}
\end{aligned}
$$

Inverse sampling approach:

- Pixels on target grid are empty
- For each pixel, apply distortion to find the best corresponding pixel in source image
- Assign interpolated values

```python
def undistort_image_inverse_sampling(image, file_name, K, distortion_params):
    h, w = image.shape[:2]
    f_x, f_y = K[0, 0], K[1, 1]
    u_0, v_0 = K[0, 2], K[1, 2]

    # empty output
    undistorted = np.zeros_like(image)
    # grid of undistorted coordinates
    u_grid, v_grid = np.meshgrid(np.arange(w), np.arange(h))
    undistorted_coords = np.stack([u_grid.ravel(), v_grid.ravel()], axis=-1)
    distorted_coords = []

    # for each coordinate, calculate corresponding distorted coordinates
    for u, v in undistorted_coords:
        # normalisation
        x = (u - u_0) / f_x
        y = (v - v_0) / f_y
        # invert distortion model
        x_dist, y_dist = x, y

        r2 = x_dist**2 + y_dist**2
        radial_distortion = 1 + distortion_params[0]*r2 + distortion_params[1]*r2**2 + distortion_params[4]*r2**3
        tangential_x = 2 * distortion_params[2] * x_dist * y_dist + distortion_params[3] * (r2 + 2 * x_dist**2)
        tangential_y = distortion_params[2] * (r2 + 2 * y_dist**2) + 2 * distortion_params[3] * x_dist * y_dist

        x_dist = x * radial_distortion + tangential_x
        y_dist = y * radial_distortion + tangential_y

        # map distorted normalized coordinates back to pixels
        u_dist = f_x * x_dist + u_0
        v_dist = f_y * y_dist + v_0
        distorted_coords.append((u_dist, v_dist))

    distorted_coords = np.array(distorted_coords).reshape(h, w, 2)
    # bilinear interpolation to sample distorted image
    map_x = distorted_coords[:, :, 0].astype(np.float32)
    map_y = distorted_coords[:, :, 1].astype(np.float32)
    undistorted = cv2.remap(image, map_x, map_y, interpolation=cv2.INTER_LINEAR, borderMode=cv2.BORDER_CONSTANT)

    os.makedirs(output_folder, exist_ok=True)
    output_path = os.path.join(output_folder, file_name)
    cv2.imwrite(output_path, undistorted)
    print(f"Saved undistorted image to {output_path}")
    return undistorted
```

Image dimensions and camera parameters

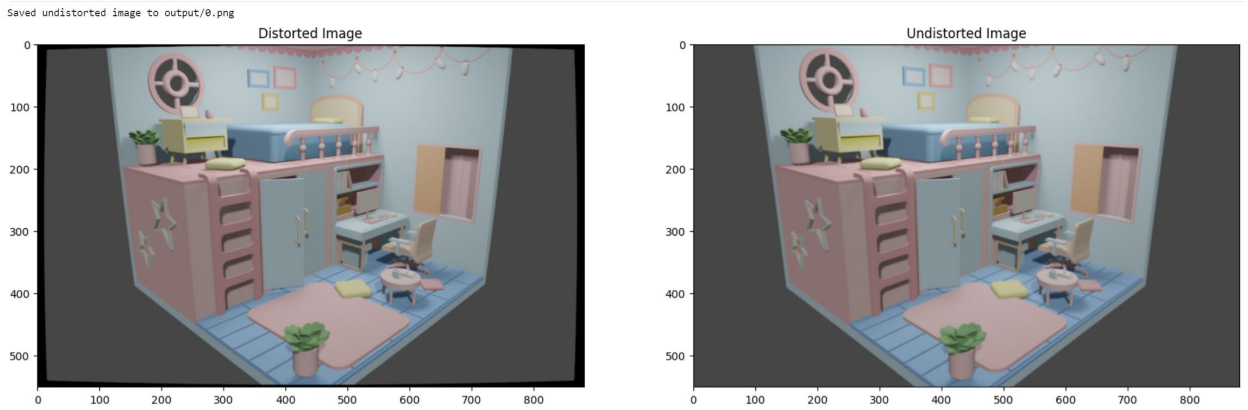
Initialise the target grid

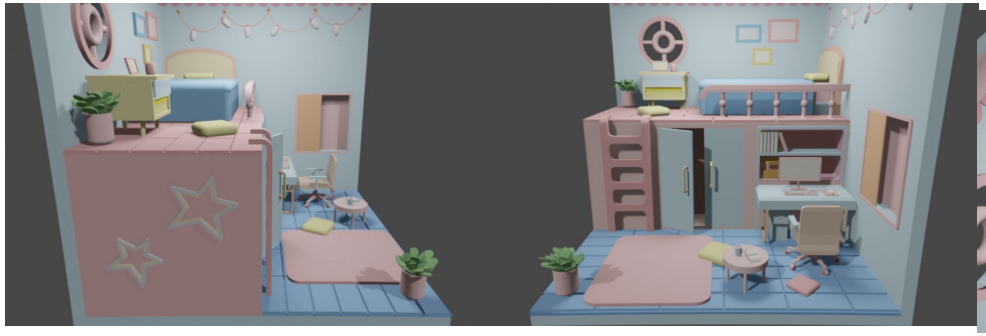
Compute distorted coordinates for each pixel

Reshape and interpolate results

Save and return image

Undistortion: Result





Feature Matching

Instance:

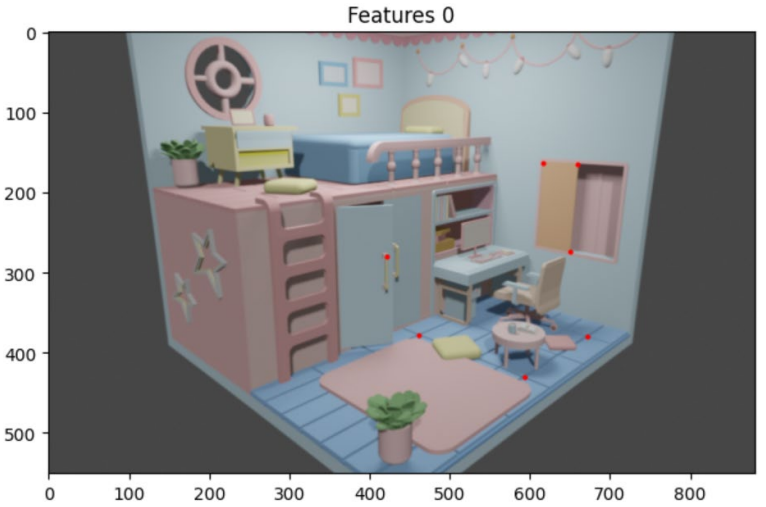
- 9 arrays with (undistorted) coordinates for (same) feature set of 7 features for each image
- 9 matrices describing transformations camera went through when taking consecutive images
- (camera parameters)

```
: print(camera_movement)

[[[ 7.07106781e-01 -1.83012702e-01  6.83012702e-01 -3.00000000e+00]
  [ 1.83012702e-01  9.80379875e-01  7.32233047e-02 -2.58819045e-01]
  [-6.83012702e-01  7.32233047e-02  7.26726907e-01  9.65925826e-01]
  [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.00000000e+00]]

 [[ 6.12323400e-17  2.58819045e-01 -9.65925826e-01  4.00000000e+00]
  [-2.58819045e-01  9.33012702e-01  2.50000000e-01 -1.03527618e+00]
  [ 9.65925826e-01  2.50000000e-01  6.69872981e-02  3.86370331e+00]
  [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  1.00000000e+00]]
```

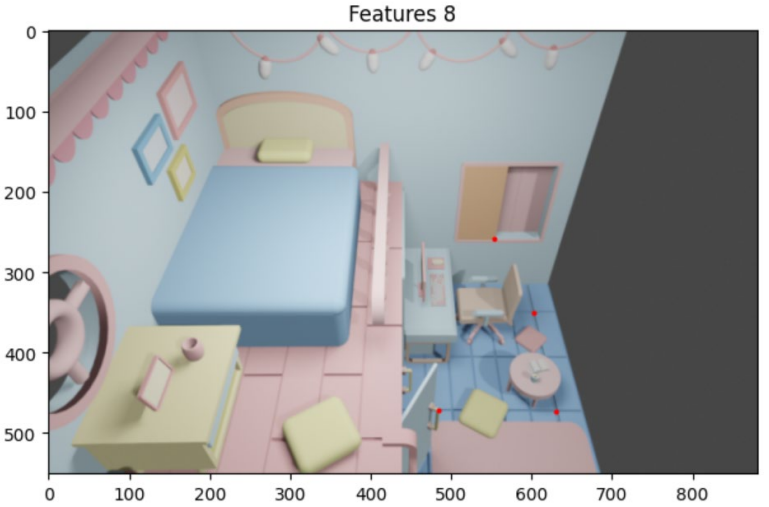
Question:

- Determine coordinates of every feature in each image (list of lists -> correspondence matrix)

```
[[[163 616]
  [431 593]
  [380 672]
  [164 660]
  [378 462]
  [280 422]
  [274 650]]
```




```
[[472 485]
 [350 602]
 [473 631]
 [259 554]
 [ -1  -1]
 [ -1  -1]
 [ -1  -1]]]
```



Feature Matching: Theory

Epipolar Geometry:

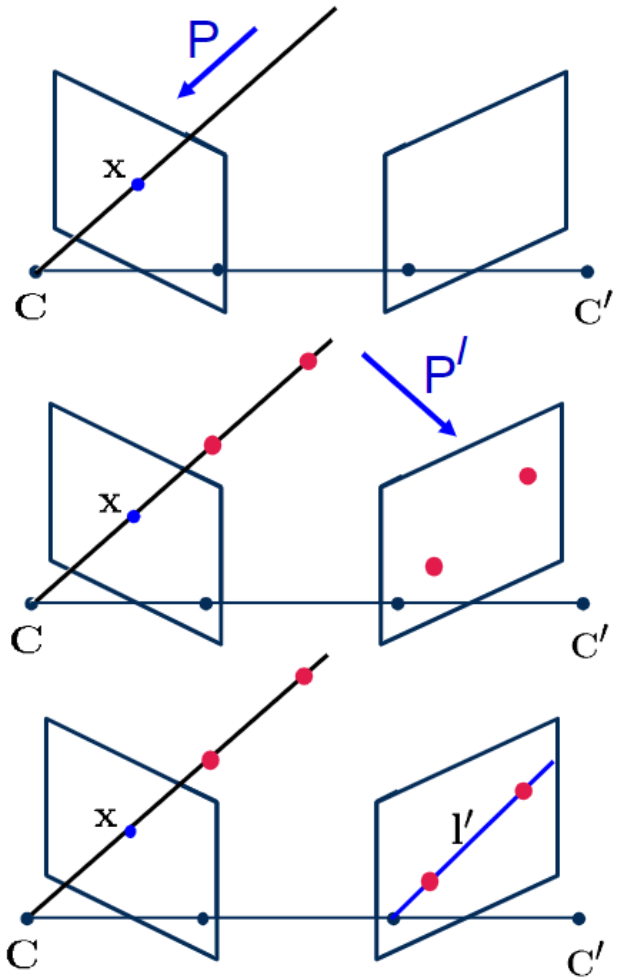
- Point *x* in one image generates an epipolar line *l’* in the second image, that the corresponding point *x’* lies on
- $l' = K^{-T}[t]_x R K^{-1} x$, where:

  K is Camera calibration matrix,

  t is Translation vector to new camera position / coordinate system,

  R is Rotation matrix to new camera position / coordinate system
- $[t]_x R = E$ is called Essential matrix
- $K^{-T}[t]_x R K^{-1} = F$ is called Fundamental matrix => *l’ = Fx*

Idea:
compute *F* -> project *l‘* -> search for the point that „almost“ lies on *l‘*



Feature Matching: Visualising Epipolar Lines

Computing fundamental matrices

```
def get_essential_matrix(R, t):
    """
    Computes essential matrix E (3x3) from rotation matrix (3x3) and translation vector (3,).
    """
    # Skew-symmetric matrix for t
    t_skew = np.array([
        [0, -t[2], t[1]],
        [t[2], 0, -t[0]],
        [-t[1], t[0], 0]
    ])
    # E = [t]_x R
    E = t_skew @ R
    return E

def get_fundamental_matrix(E, K):
    """
    Computes fundamental matrix F (3x3) from essential matrix E (3x3) and camera matrix K (3x3).
    """
    # F = K^-T E K^-1
    F = np.linalg.inv(K).T @ E @ np.linalg.inv(K)
    # Normalisation
    F = F / F[2, 2]
    return F
```

Function to compute Essential matrix

Function to compute Fundamental matrix

```
num_images = len(undistorted_images)

F = []
for i in range(num_images-1):
    R = camera_movement[i][:3, :3]
    t = camera_movement[i][:3, 3]
    E = get_essential_matrix(R, t)
    F.append(get_fundamental_matrix(E, K))
```

Now do this

Projecting epipolar lines

To iterate and distinct

```
colors = ['r', 'g', 'b', 'c', 'm', 'y', 'k'] # Matplotlib color codes
```

Pairs of consecutive images

```
for i in range(num_images-1):
    img1 = cv2.cvtColor(cv2.imread(undistorted_images[i]), cv2.COLOR_BGR2RGB)
    img2 = cv2.cvtColor(cv2.imread(undistorted_images[i+1]), cv2.COLOR_BGR2RGB)

    fig, axs = plt.subplots(1, 2, figsize=(10, 5))
    # Features in first image
    axs[0].imshow(img1)
    k = 0
    for p1 in feature_data[i]:
        if p1[0] == -1:
            continue
        axs[0].plot(p1[1], p1[0], colors[k] + 'o', markersize=3) # height is x, width is y
        k += 1
```

Plot features in colour in the first image

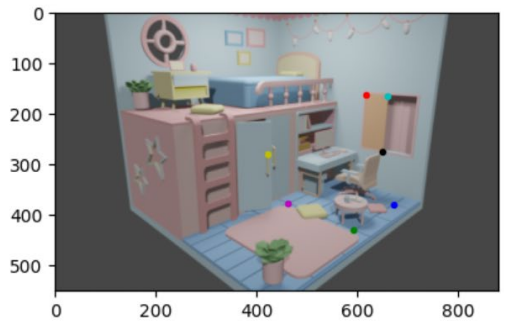
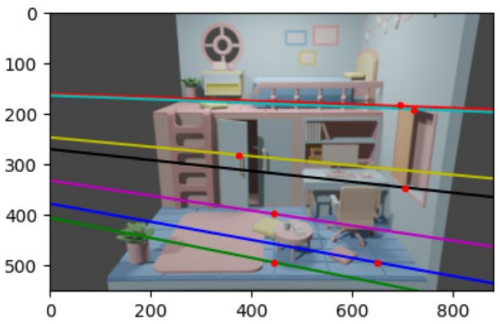
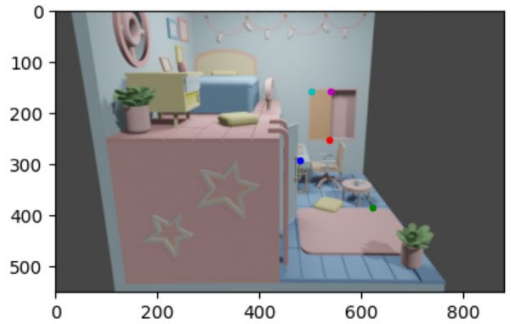
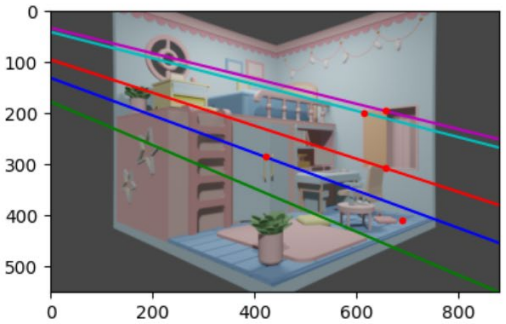
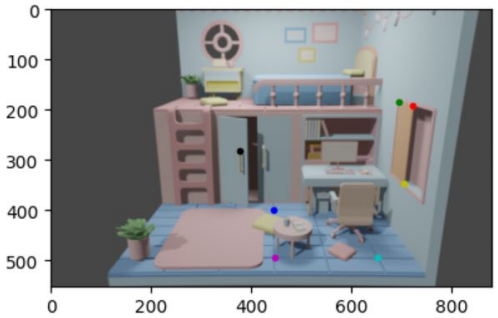
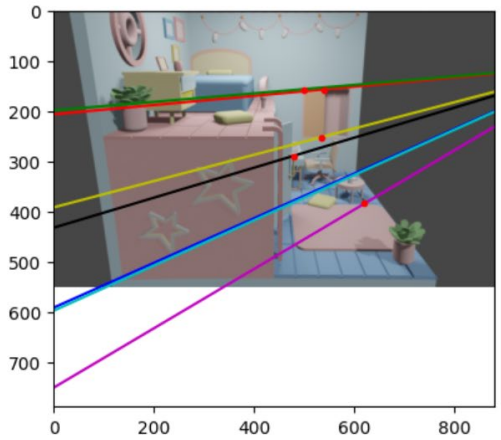
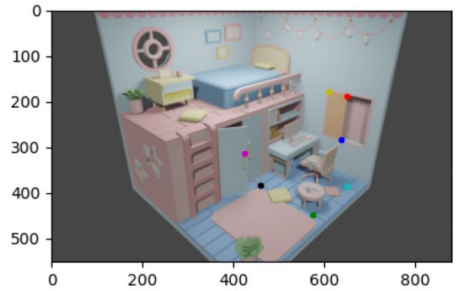
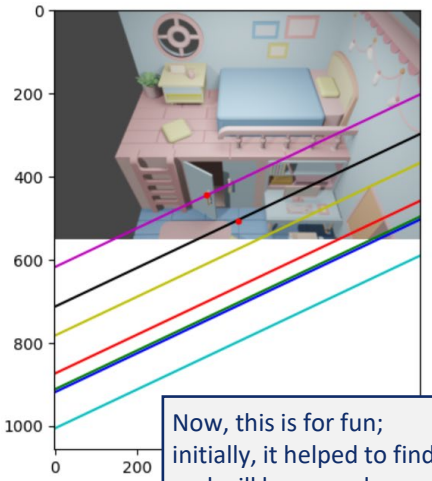
```
    axs[1].imshow(img2)
    # Epipolar lines from features in second image using respective colours
    k = 0
    for p1 in feature_data[i]:
        if p1[0] == -1:
            continue
        x1 = np.array([p1[1], p1[0], 1])  # homogeneous coordinates
        l2 = F[i] @ x1  # epipolar line
        l2 = [l2[0]/l2[2], l2[1]/l2[2], 1.0]

        h, w = img2.shape[:2]
        x0, y0 = 0, -l2[2] / l2[1]
        x1, y1 = h, -(l2[2] + l2[0] * h) / l2[1]
        axs[1].axline((x1, y1), (x0, y0), color=colors[k])
        k += 1
```

Plot epipolar lines in corresponding colours in the second image

```
    # Features in second image
    for p1 in feature_data[i+1]:
        if p1[0] == -1:
            continue
        axs[1].plot(p1[1], p1[0], 'ro', markersize=3)

    plt.show()
```

Just plot features of the second image

Feature Matching: Visualising Epipolar Lines

...

Now, this is for fun;
initially, it helped to find and fix a lot of bugs;
and will be reused soon

Feature Matching: Implementation for Consecutive Images

```
num_features = len(feature_data[0])

def match_feature_(feature, features_to_match, F, distance_threshold=2):
    """
    Matches feature to that of another view.
    """
    if feature[0] == -1:
        return -1

    x1 = np.array([feature[1], feature[0], 1])
    epipolar_line = F @ x1
    epipolar_line = [epipolar_line[0]/epipolar_line[2], epipolar_line[1]/epipolar_line[2], 1.0]
    best_match = -1
    min_distance = float('inf')

    for l in range(len(features_to_match)):
        if features_to_match[l, 0] != -1:
            x2 = np.array([features_to_match[l, 1], features_to_match[l, 0], 1])
            distance = np.abs(epipolar_line[0] * x2[0] + epipolar_line[1] * x2[1] + epipolar_line[2]) / np.sqrt(epipolar_line[0]**2 + epipolar_line[1]**2)
            if distance < min_distance:
                min_distance = distance
                best_match = l

    if min_distance < distance_threshold:
        return best_match
    return -1

matches = []
for i in range(num_images-1):
    matches_i = []
    for j in range(num_features):
        match = match_feature_(feature_data[i, j], feature_data[i+1], F[i])
        if match != -1:
            matches_i.append((j, match))
    matches.append(matches_i)
    print(f"{i}->{i+1}", matches_i)
```

```
0->1 [(0, 1), (1, 4), (2, 3), (3, 0), (4, 2), (5, 6), (6, 5)]
1->2 [(0, 6), (1, 3), (4, 1), (5, 0), (6, 2)]
2->3 [(0, 3), (2, 6), (3, 1), (6, 4)]
3->4 [(1, 0), (3, 5), (4, 4), (5, 1), (6, 3)]
4->5 [(0, 1), (3, 0), (4, 4), (5, 6)]
5->6 [(0, 4), (1, 5), (4, 0), (6, 2)]
6->7 [(4, 4), (6, 6)]
7->8 [(4, 0)]
```

Compute epipolar line

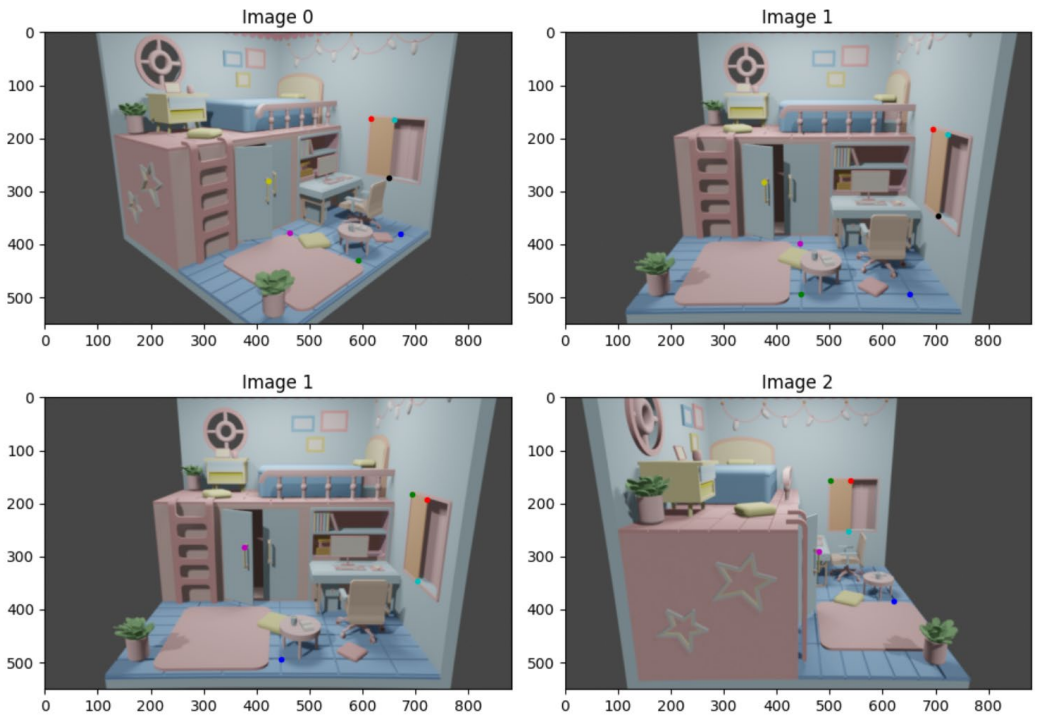
Find closest match

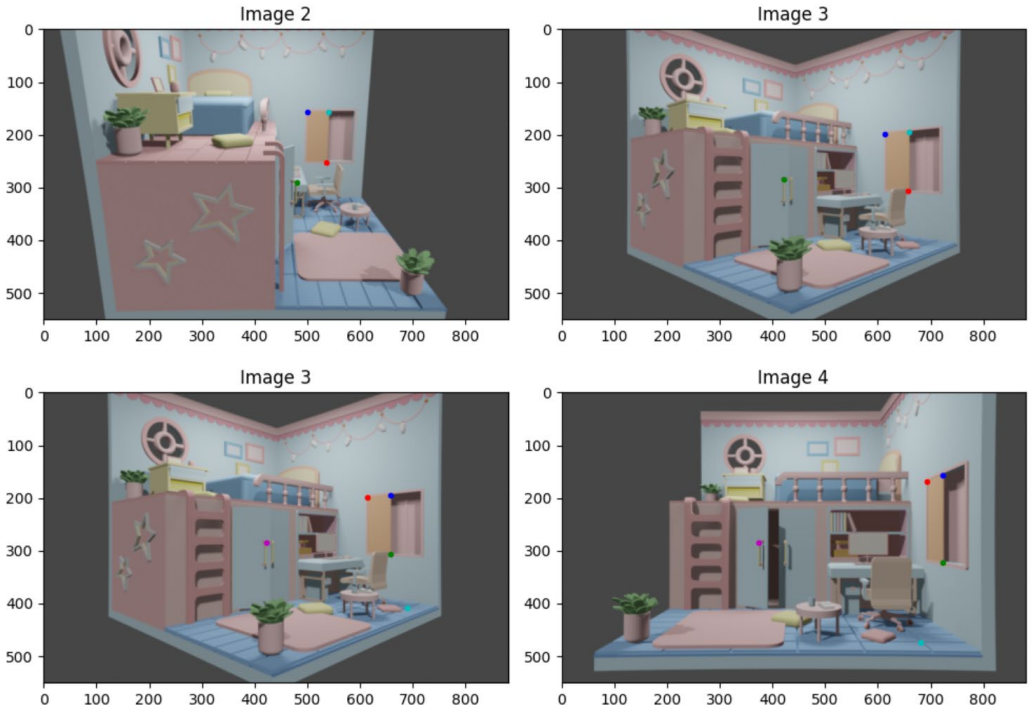
... within threshold parameter

Matches between consecutive images

- But is this sufficient?
- Can there be at least one complete chain?

Feature Matching: Implementation for Consecutive Images





...

Feature Matching: Accumulating Algorithm

Idea (we are solving missing features problem):

- Take yet unknown feature and find the correspondences from view 0 (w.l.o.g.) in all other views, where it is possible
- If feature is not found in a view… there are still other views, apart from view 0, where it now has been found to project from

  -> accumulating algorithm

But first:

- Just have a look on all possible epipolar lines projections…
  - Repeat plotting for all pairs of views

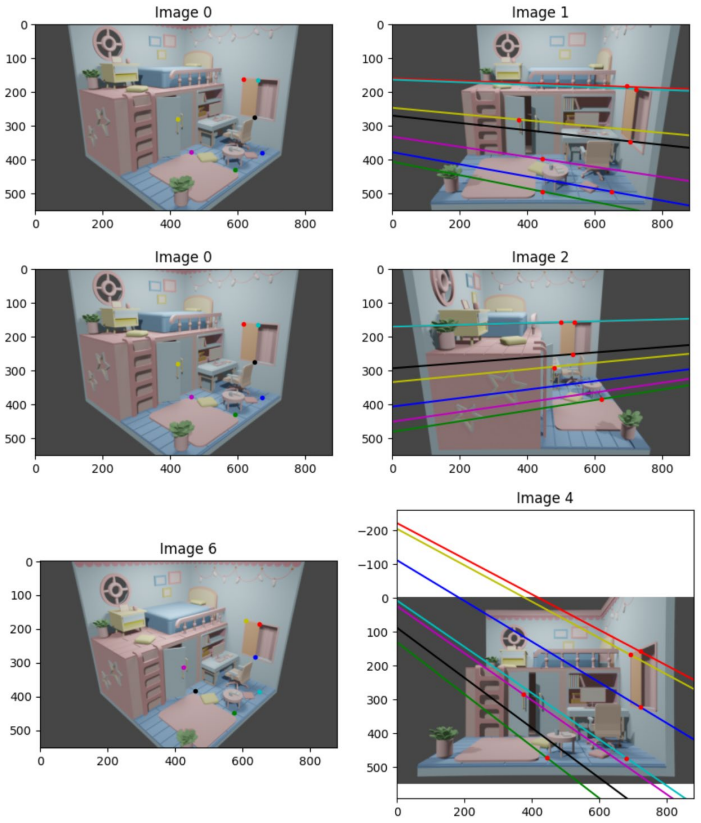
One more handy function

```python
def get_transformation_matrices(view_idx):
    """
    Calculates transformation matrices from the given view to all views
    """
    pose = np.eye(4)
    transformations = [pose]
    for i in range(num_images-1 - view_idx):
        pose = camera_movement[i+view_idx] @ pose
        transformations.append(pose)

    pose = np.eye(4)
    for i in range(view_idx):
        pose = np.linalg.inv(camera_movement[view_idx-1-i]) @ pose
        transformations.insert(0, pose)
    return transformations
```

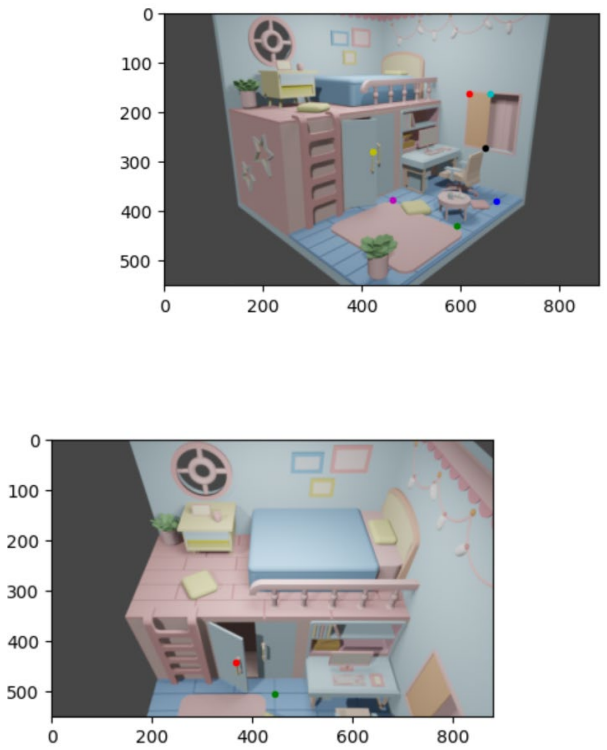
Array of transformation matrices from the given view to all views including itself, starting from view 0

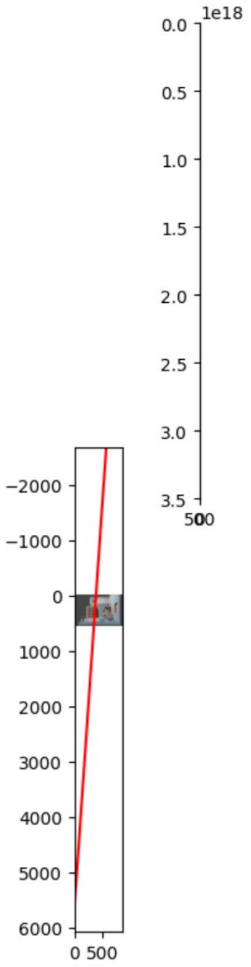
Feature Matching: Projecting *All* Epipolar Lines

fine

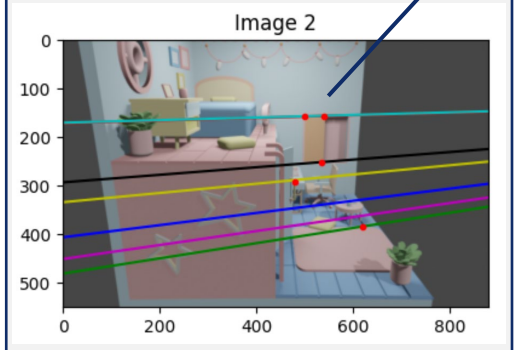


What is this?



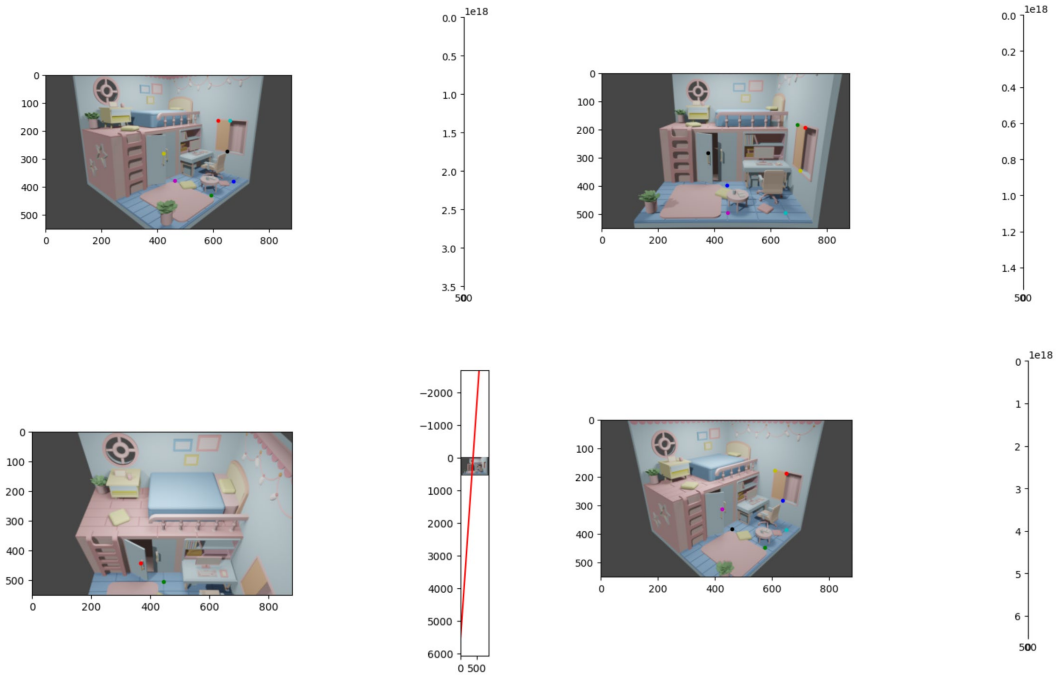


And for now, let us also admit that this one is trying to cause problems



Feature Matching: Inaccuracy on Similar Views

This happens between images e.g. 0 and 3, 1 and 7, 7 and 4, 6 and 3 …



On debugging, it can be seen that camera positions are too close to each other per some axes, generating too close epipolar lines

This also leads to inaccurate feature matching between such camera poses due to float arithmetics

Feature Matching: Accumulating Algorithm

Idea:

- Continue with accumulating algorithm
- Detect and just skip „bad views“
- Cover skipped features with other views

feature_correspondence_matrix[][]

Feature	1	2	3	...	7
In view 0					
In view 1					
...					
In view 8					

Building correspondence matrix

```
feature_correspondence_matrix = []
features_matched = []

for i in range(num_images):
    feature = []
    matches = []
    for j in range(num_features):
        feature.append([-1, -1])
        matches.append(False)
    feature_correspondence_matrix.append(feature)
    features_matched.append(matches)

for i in range(num_features):
    feature_correspondence_matrix[0][i] = feature_data[0][i]
    features_matched[0][i] = True
```

Track if features are matched (and if there is no correct match)

Initialisation from the first image

Iteratively fill missing correspondences

```
done = True
for i in range(num_images):
    find_missing_correspondences(i)
    done = True
    for k in range(num_images):
        for j in range(num_features):
            if features_matched[k][j] == False:
                done = False
                break
    if done:
        break

for i in range(len(feature_correspondence_matrix)):
    print(feature_correspondence_matrix[i])
```

```
def find_missing_correspondences(view_idx):
    transformations = get_transformation_matrices(view_idx)
    for i in range(num_images):
        if i == view_idx:
            continue

        images_matched = True
        for j in range(num_features):
            if features_matched[i][j] == False:
                images_matched = False
        if images_matched:
            continue

        R = transformations[i][:3, :3]
        t = transformations[i][:3, 3]
        E = get_essential_matrix(R, t)
        F = get_fundamental_matrix(E, K)

        correspondences = []
        matched = []
        bad_view = False

        for j in range(num_features):
            feature = match_feature(feature_correspondence_matrix[view_idx][j], feature_data[i], F)
            if feature[0] == 0 and feature[1][0] == -1:
                bad_view = True
                break
            elif feature[0] == 0:
                correspondences.append([-1, -1])
                matched.append(False)
            else:
                correspondences.append(feature[1])
                matched.append(True)

        if bad_view:
            continue

        for j in range(num_features):
            if features_matched[i][j] == False:
                feature_correspondence_matrix[i][j] = correspondences[j]
                features_matched[i][j] = matched[j]
```

Handy function comes in use

Iterate through all images except the given one

Skip if the views already were matched

Compute E and F

Match!

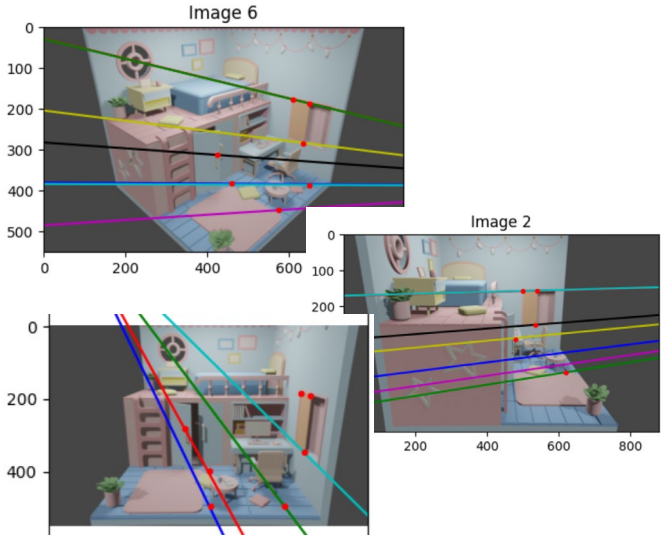
Modified function is to be explained

Skip if it is “bad” view

Update!

Feature Matching

These do result in incorrect matches:



Idea:

- “Ambiguity” parameter

```python
h, w = img2.shape[:2]
ambig_param = 1.1

def match_feature(feature, features_to_match, F, distance_threshold=5):
    """
    Matches feature to that of another view, or returns (1, [-1, -1]) if feature was not found, (0, [-1, -1]) and (0, [0, 0]) in case of bad view or ambiguity respectively.
    """
    if feature[0] == -1:
        return (1, [-1, -1])

    x1 = np.array([feature[1], feature[0], 1])
    epipolar_line = F @ x1
    epipolar_line = [epipolar_line[0]/epipolar_line[2], epipolar_line[1]/epipolar_line[2], 1.0]

    x0, y0 = 0, -epipolar_line[2] / epipolar_line[1]
    x1, y1 = h, -(epipolar_line[2] + epipolar_line[0] * h) / epipolar_line[1]
    if abs(y0) > (h+w) * 2 or abs(y1) > (h+w) * 2:
        return (0, [-1, -1])

    best_match = -1
    min_distance = float('inf')
    ambiguity = False

    for l in range(len(features_to_match)):
        if features_to_match[l, 0] != -1:
            x2 = np.array([features_to_match[l, 1], features_to_match[l, 0], 1])
            distance = np.abs(epipolar_line[0] * x2[0] + epipolar_line[1] * x2[1] + epipolar_line[2]) / np.sqrt(epipolar_line[0]**2 + epipolar_line[1]**2)
            if abs(distance-min_distance) < ambig_param:
                ambiguity = True

            if distance < min_distance:
                if abs(distance-min_distance) >= ambig_param:
                    ambiguity = False
                min_distance = distance
                best_match = l

    if ambiguity:
        return (0, [0, 0])
    if min_distance < distance_threshold:
        return (1, features_to_match[best_match])

    return (1, [-1, -1])
```

Compute epipolar line

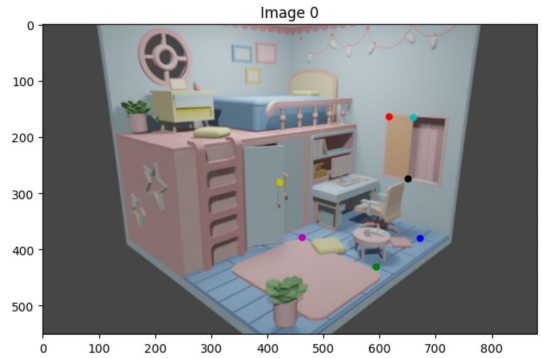
Detect “bad” view: epipolar line is not / hardly visible

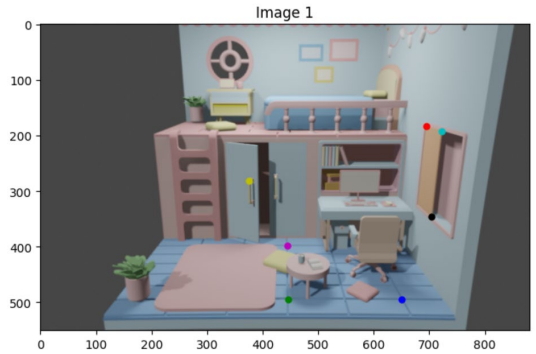
Iterate through features to find the best match

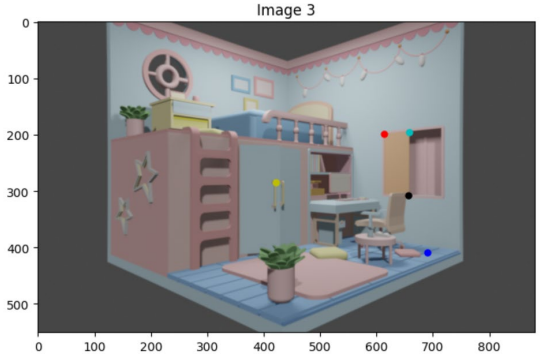
Detect “ambiguity”: another feature is also too close

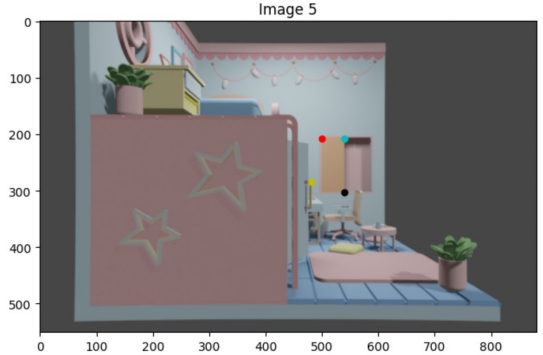
1. **Feature not found** (returns `(1, [-1, -1])`).
2. **Bad view** (returns `(0, [-1, -1])`).
3. **Ambiguity** (returns `(0, [0, 0])`).
4. **Successful match** (returns `(1, coordinates_of_best_match)`).

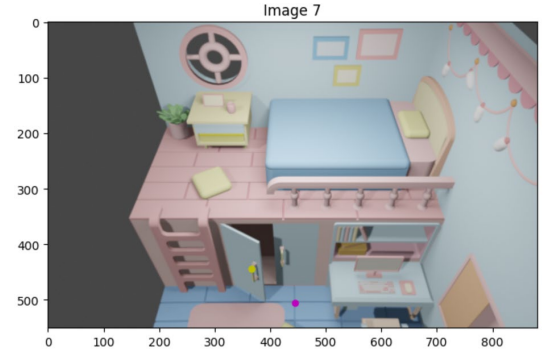
Feature Matching: Result

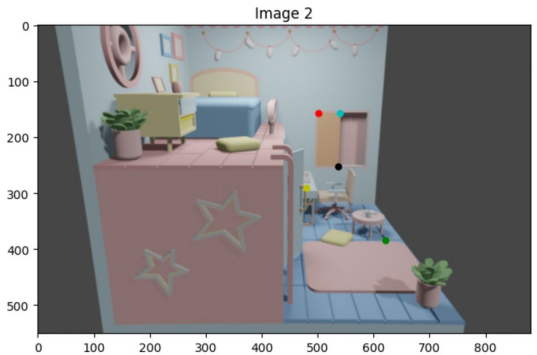


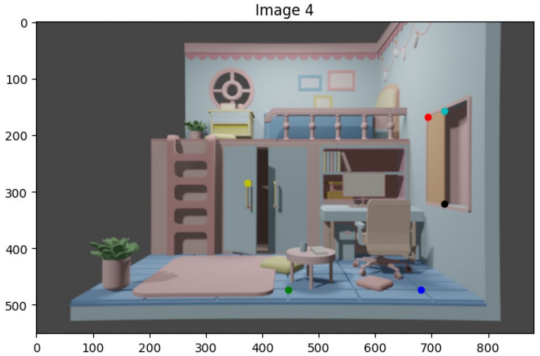


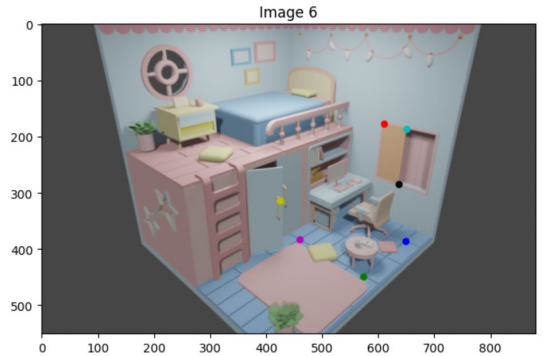


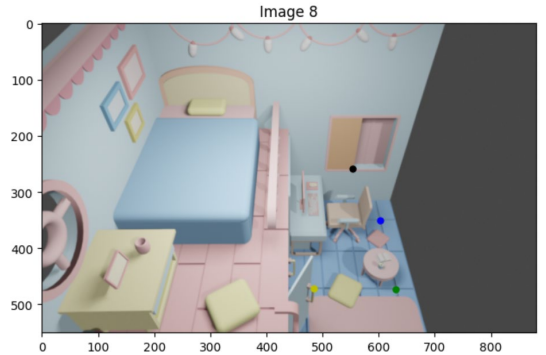












Feature Matching Bonus: See Iterative-ness

```
iteration 0
[array([163, 616], dtype=int32), array([431, 593], dtype=int32), array([380, 672], dtype=int32), array([164, 660], dtype=int32), array([378, 462], dtype=int32), array([280, 422], dtype=int32), array([27
4, 650], dtype=int32)]
[array([183, 694], dtype=int32), array([495, 446], dtype=int32), array([495, 651], dtype=int32), array([192, 722], dtype=int32), array([398, 444], dtype=int32), array([282, 376], dtype=int32), array([34
6, 704], dtype=int32)]
[[-1, -1], array([384, 621], dtype=int32), [-1, -1], [-1, -1], [-1, -1], array([291, 480], dtype=int32), array([252, 536], dtype=int32)]
[[-1, -1], [-1, -1], [-1, -1], [-1, -1], [-1, -1], [-1, -1], [-1, -1]]
[array([168, 693], dtype=int32), array([473, 446], dtype=int32), array([474, 681], dtype=int32), array([157, 722], dtype=int32), [-1, -1], array([285, 374], dtype=int32), array([322, 722], dtype=int32)]
[array([207, 500], dtype=int32), [-1, -1], [-1, -1], array([207, 539], dtype=int32), [-1, -1], array([285, 481], dtype=int32), array([303, 539], dtype=int32)]
[[-1, -1], [-1, -1], [-1, -1], [-1, -1], [-1, -1], [-1, -1], [-1, -1]]
[[-1, -1], [-1, -1], [-1, -1], [-1, -1], array([505, 444], dtype=int32), array([444, 367], dtype=int32), [-1, -1]]
[[-1, -1], array([473, 631], dtype=int32), array([350, 602], dtype=int32), [-1, -1], [-1, -1], array([472, 485], dtype=int32), array([259, 554], dtype=int32)]
iteration 1
[array([163, 616], dtype=int32), array([431, 593], dtype=int32), array([380, 672], dtype=int32), array([164, 660], dtype=int32), array([378, 462], dtype=int32), array([280, 422], dtype=int32), array([27
4, 650], dtype=int32)]
[array([183, 694], dtype=int32), array([495, 446], dtype=int32), array([495, 651], dtype=int32), array([192, 722], dtype=int32), array([398, 444], dtype=int32), array([282, 376], dtype=int32), array([34
6, 704], dtype=int32)]
[array([157, 501], dtype=int32), array([384, 621], dtype=int32), [-1, -1], array([157, 540], dtype=int32), [-1, -1], array([291, 480], dtype=int32), array([252, 536], dtype=int32)]
[array([199, 614], dtype=int32), [-1, -1], array([409, 690], dtype=int32), array([195, 658], dtype=int32), [-1, -1], array([285, 422], dtype=int32), array([307, 657], dtype=int32)]
[array([168, 693], dtype=int32), array([473, 446], dtype=int32), array([474, 681], dtype=int32), array([157, 722], dtype=int32), [-1, -1], array([285, 374], dtype=int32), array([322, 722], dtype=int32)]
[array([207, 500], dtype=int32), [-1, -1], [-1, -1], array([207, 539], dtype=int32), [-1, -1], array([285, 481], dtype=int32), array([303, 539], dtype=int32)]
[[-1, -1], array([448, 574], dtype=int32), array([386, 649], dtype=int32), [-1, -1], array([383, 460], dtype=int32), array([313, 424], dtype=int32), array([284, 636], dtype=int32)]
[[-1, -1], [-1, -1], [-1, -1], [-1, -1], array([505, 444], dtype=int32), array([444, 367], dtype=int32), [-1, -1]]
[[-1, -1], array([473, 631], dtype=int32), array([350, 602], dtype=int32), [-1, -1], [-1, -1], array([472, 485], dtype=int32), array([259, 554], dtype=int32)]
iteration 2
[array([163, 616], dtype=int32), array([431, 593], dtype=int32), array([380, 672], dtype=int32), array([164, 660], dtype=int32), array([378, 462], dtype=int32), array([280, 422], dtype=int32), array([27
4, 650], dtype=int32)]
[array([183, 694], dtype=int32), array([495, 446], dtype=int32), array([495, 651], dtype=int32), array([192, 722], dtype=int32), array([398, 444], dtype=int32), array([282, 376], dtype=int32), array([34
6, 704], dtype=int32)]
[array([157, 501], dtype=int32), array([384, 621], dtype=int32), [-1, -1], array([157, 540], dtype=int32), [-1, -1], array([291, 480], dtype=int32), array([252, 536], dtype=int32)]
[array([199, 614], dtype=int32), [-1, -1], array([409, 690], dtype=int32), array([195, 658], dtype=int32), [-1, -1], array([285, 422], dtype=int32), array([307, 657], dtype=int32)]
[array([168, 693], dtype=int32), array([473, 446], dtype=int32), array([474, 681], dtype=int32), array([157, 722], dtype=int32), [-1, -1], array([285, 374], dtype=int32), array([322, 722], dtype=int32)]
[array([207, 500], dtype=int32), [-1, -1], [-1, -1], array([207, 539], dtype=int32), [-1, -1], array([285, 481], dtype=int32), array([303, 539], dtype=int32)]
[array([177, 610], dtype=int32), array([448, 574], dtype=int32), array([386, 649], dtype=int32), array([187, 650], dtype=int32), array([383, 460], dtype=int32), array([313, 424], dtype=int32), array([28
4, 636], dtype=int32)]
[[-1, -1], [-1, -1], [-1, -1], [-1, -1], array([505, 444], dtype=int32), array([444, 367], dtype=int32), [-1, -1]]
[[-1, -1], array([473, 631], dtype=int32), array([350, 602], dtype=int32), [-1, -1], [-1, -1], array([472, 485], dtype=int32), array([259, 554], dtype=int32)]
```

e.g. “bad” views between 0 and 3

We were done after the the second (third) iteration

Triangulation

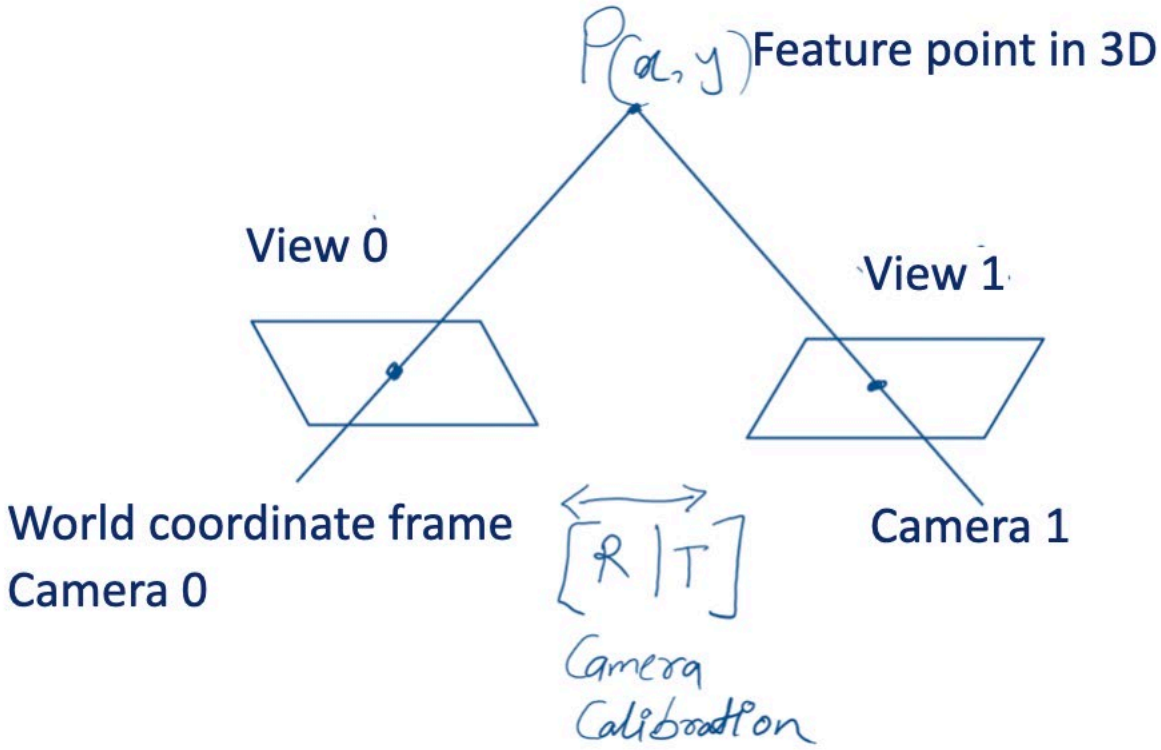
Estimating the 3D position of a point by analyzing two or more 2D images taken from different viewpoints

We need:

K - Camera calibration matrix for each view

R | T - Transformation matrix for each view wrt world camera

Feature point coordinates from each view.



Triangulation

Procedure -

Given a feature in n views, get its 3D position.

1. Select coordinates of the Feature in the views it is seen.
2. Create the A matrix:
   a. x . P[2] - P[0] = 0 y . P[1] - P[0] = 0
   b. Do this for all x, y and P in each view and concatenate.
3. Compute the SVD of A.
4. The last column of the last matrix i.e., V is the point with homogeneous coordinate.
5. Normalize by dividing with the homogeneous coordinate.

```python
def multiview_triangulation(features_matrix, projection_matrices):
    triangulated_points = []
    for feature_index, feature in enumerate(features_matrix):
        A = []

        for view_index, point in enumerate(feature):
            if np.array(point).sum() == -2:
                continue
            P = projection_matrices[view_index]
            #since points are (h,w) .ie (y,x)
            y, x = point

            A.append(x * P[2, :] - P[0, :])  # Row from x-coordinate
            A.append(y * P[2, :] - P[1, :])  # Row from y-coordinate

        A = np.array(A)  # Convert to NumPy array


        # Solve A * X = 0 using SVD
        _, _, Vt = np.linalg.svd(A)
        X_homogeneous = Vt[-1]  # Last row of Vt corresponds to small

        # Convert to Euclidean coordinates
        X_euclidean = X_homogeneous[:-1] / X_homogeneous[-1]
        triangulated_points.append(X_euclidean)
```

Triangulation

Result - 3D coordinates of all feature points

```
Feature 0: [ 1.35201537 -0.86230485  4.81112929]
Feature 1: [0.90839178 0.92869221 3.73654327]
Feature 2: [1.63211365 0.744573   4.43802616]
Feature 3: [ 1.60153403 -0.8015058   4.58633108]
Feature 4: [0.1593758  0.72950708 4.45851857]
Feature 5: [-0.11231227  0.03941065  3.93977091]
Feature 6: [ 1.59977506e+00 -4.41454716e-03  4.80063840e+00]
```

One-View Reconstruction

We have 3D coordinates for feature points with world coordinate system as view 0 => Z-coordinates can be related with depth values

- Use simple proportion to calculate depth Z of a point
- $$\begin{pmatrix} x_{pix} \\ y_{pix} \\ 1 \end{pmatrix} = K[I|0_3] \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$
  $$K[I|0_3] = \begin{pmatrix} f & 0 & c_y & 0 \\ 0 & f & c_x & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$
  $$Z = 1:$$
  $$fX + c_y Z = x_{pix} \iff X = \frac{x_{pix} - c_y}{f}$$
  $$fY + c_x Z = y_{pix} \iff Y = \frac{y_{pix} - c_x}{f}$$
- Multiply by actual Z => find X and Y

Relate depth values to Z-coordinate

```
def get_depth_for_features(feature_correspondence_matrix, depth_maps):
    depth_values = []

    for image_index, feature_points in enumerate(feature_correspondence_matrix):
        image_depth_values = []
        depth_map = depth_maps[image_index]

        for point in feature_points:
            if np.array_equal(point, [-1, -1]):
                image_depth_values.append(None)
            else:
                u, v = point
                depth_value = depth_map[u, v]
                image_depth_values.append(depth_value)

        depth_values.append(image_depth_values)

    return depth_values

depth_values = get_depth_for_features(feature_correspondence_matrix, depth_maps)

for image_index, image_depth_values in enumerate(depth_values):
    print(f"Image {image_index}:")
    for feature_index, depth in enumerate(image_depth_values):
        print(f"  Feature {feature_index}: Depth (0-255 scale) = {depth}")
```

```
Image 0:
  Feature 0: Depth (0-255 scale) = 204
  Feature 1: Depth (0-255 scale) = 160
  Feature 2: Depth (0-255 scale) = 190
  Feature 3: Depth (0-255 scale) = 193
  Feature 4: Depth (0-255 scale) = 190
  Feature 5: Depth (0-255 scale) = 167
  Feature 6: Depth (0-255 scale) = 204
```

```
Feature 0: [ 1.35201537 -0.86230485  4.81112929]
Feature 1: [0.90839178 0.92869221 3.73654327]
Feature 2: [1.63211365 0.744573   4.43802616]
Feature 3: [ 1.60153403 -0.8015058   4.58633108]
Feature 4: [0.1593758  0.72950708 4.45851857]
Feature 5: [-0.11231227  0.03941065  3.93977091]
Feature 6: [ 1.59977506e+00 -4.41454716e-03  4.80063840e+00]
```

One-View Reconstruction

Generate pixel coordinate datasets for point cloud

```
dataset_height = range(1, 551, 2)
dataset_width = range(1, 881, 2)
dataset_coordinates = []

for i in range(len(dataset_height)):
    for j in range(len(dataset_width)):
        dataset_coordinates.append([dataset_height[i], dataset_width[j]])

#print(dataset_coordinates)
print(len(dataset_coordinates))
```

```
121000
```

Calculate 3D coordinates for the generated point dataset

for view 0

```
def get_depths(points):
    depths = []
    for i in range(len(points)):
        pixel_depth = get_point_depth_in_view(points[i], 0)
        if pixel_depth == 255 or pixel_depth == 0:
            depths.append(-1)
        else:
            depth = points_3d[0][2] * pixel_depth / get_point_depth_in_view(feature_correspondence_matrix[0][0], 0)
            depths.append(depth)
    return depths

dataset_z = get_depths(dataset_coordinates)
#print(dataset_z)
```

```
dataset_3d = []
for i in range(len(dataset_coordinates)):
    z = dataset_z[i]
    if (z == -1):
        continue

    x = (dataset_coordinates[i][0] - principal_point[1]) / focal_length_px * z
    y = (dataset_coordinates[i][1] - principal_point[0]) / focal_length_px * z
    dataset_3d.append([x, y, z])
```

```
def get_point_depth_in_view(point, view_idx):
    depth_map = depth_maps[view_idx]
    depth_value = depth_map[point[0], point[1]]
    return depth_value
```

Extract colours for the selected points

```
def get_point_colour_in_view(point, view_idx):
    colour_value = images[0][point[0]][point[1]]
    return colour_value
```

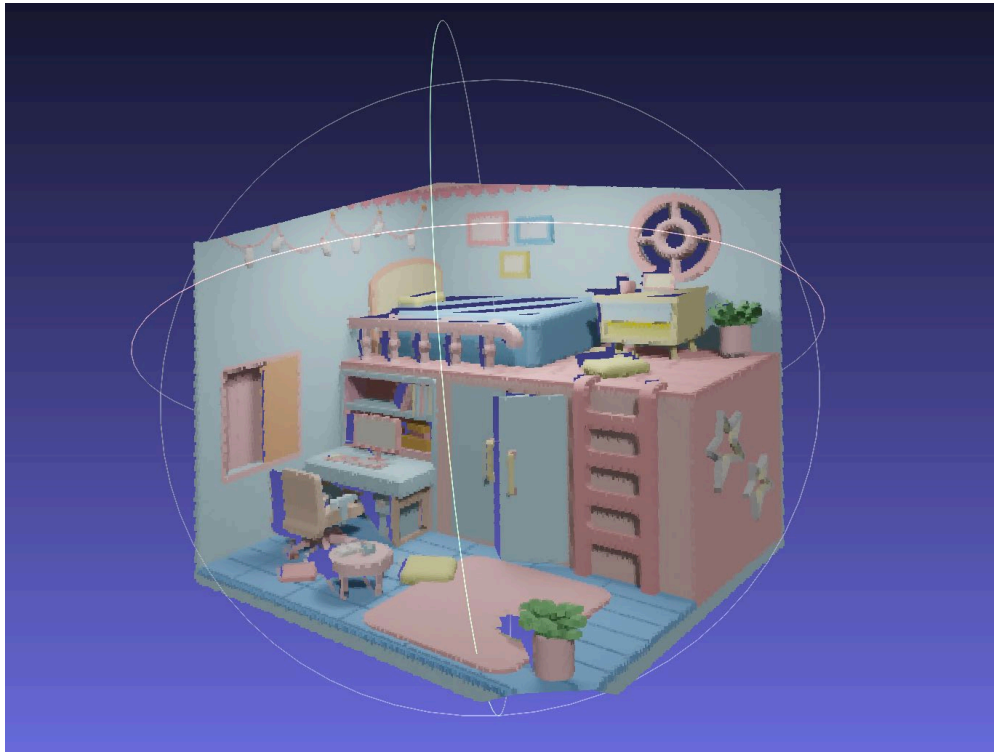
```
dataset_colours = []
for i in range(len(dataset_coordinates)):
    z = dataset_z[i]
    if (z == -1):
        continue
    colour = get_point_colour_in_view(dataset_coordinates[i], 0)
    dataset_colours.append(colour)
```

Get mesh via utility function!

```
def ply_creator(input_3d, rgb_data, filename):
    assert (input_3d.ndim == 2), "Pass 3d points as NumPointsX3 array "
    text1 = """ply\nformat ascii 1.0"""
    text2 = "element vertex " + str(input_3d.shape[0])
    text3 = """property float x
    property float y
    property float z
    property uchar red
    property uchar green
    property uchar blue
    end_header"""

    with open(filename, 'w') as ply_file:
        ply_file.write(text1 + "\n")
        ply_file.write(text2 + "\n")
        ply_file.write(text3 + "\n")
        for i in range(input_3d.shape[0]):
            ply_file.write(f"{input_3d[i, 0]} {input_3d[i, 1]} {input_3d[i, 2]} {rgb_data[i][2]} {rgb_data[i][1]} {rgb_data[i][0]}\n")
```

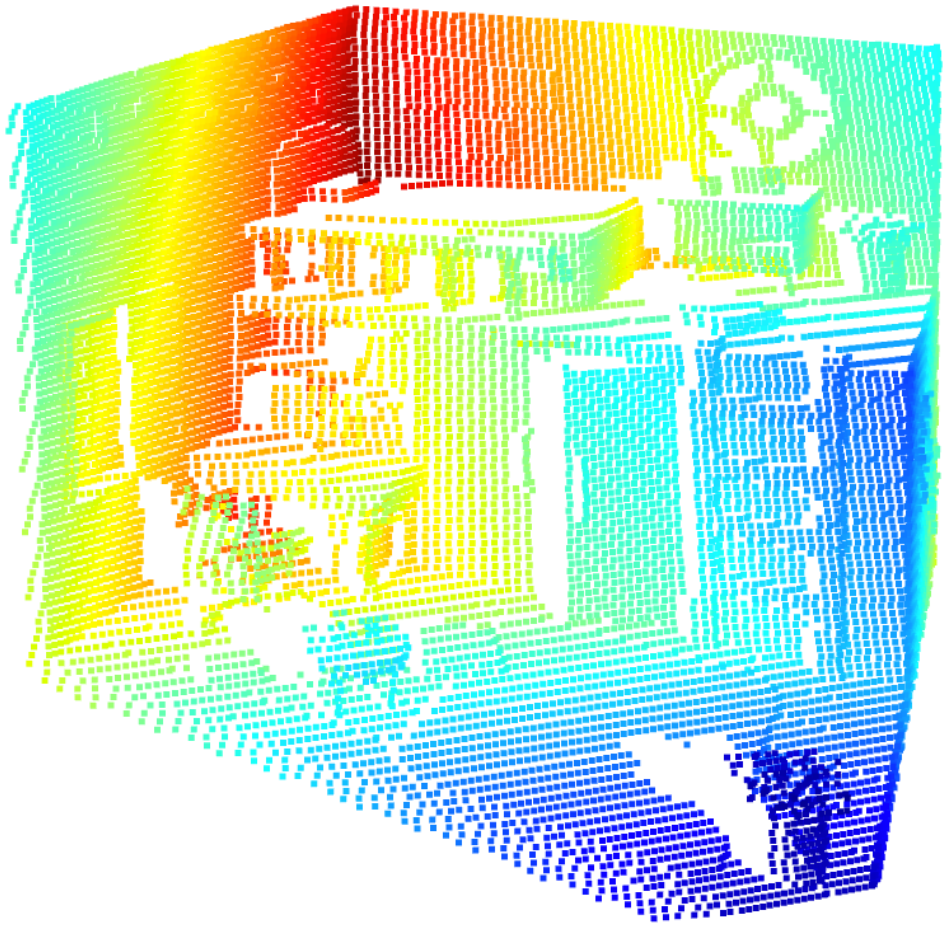
```
dataset_3d = np.array(dataset_3d)
dataset_colours = np.array(dataset_colours)
ply_creator(dataset_3d, dataset_colours, 'sparse_reconstruction.ply')
```

One-View Reconstruction: Result



Sparse Reconstruction

- **2D Grid Points**: Created a grid of 2D points with defined height and width steps.
- **Reference Feature**: Used the first feature from the correspondence matrix as a reference.
- **Depth Calculation**: Computed depths for grid points based on the reference feature and depth map.
- **3D Point Generation**: Converted 2D points and depths into 3D coordinates using camera intrinsics.
- **Sparse Point Cloud**: Created and visualized a sparse point cloud in Open3D.



Load Depth Maps, Calibration data and Camera Poses

Depth Maps and RGB Images:

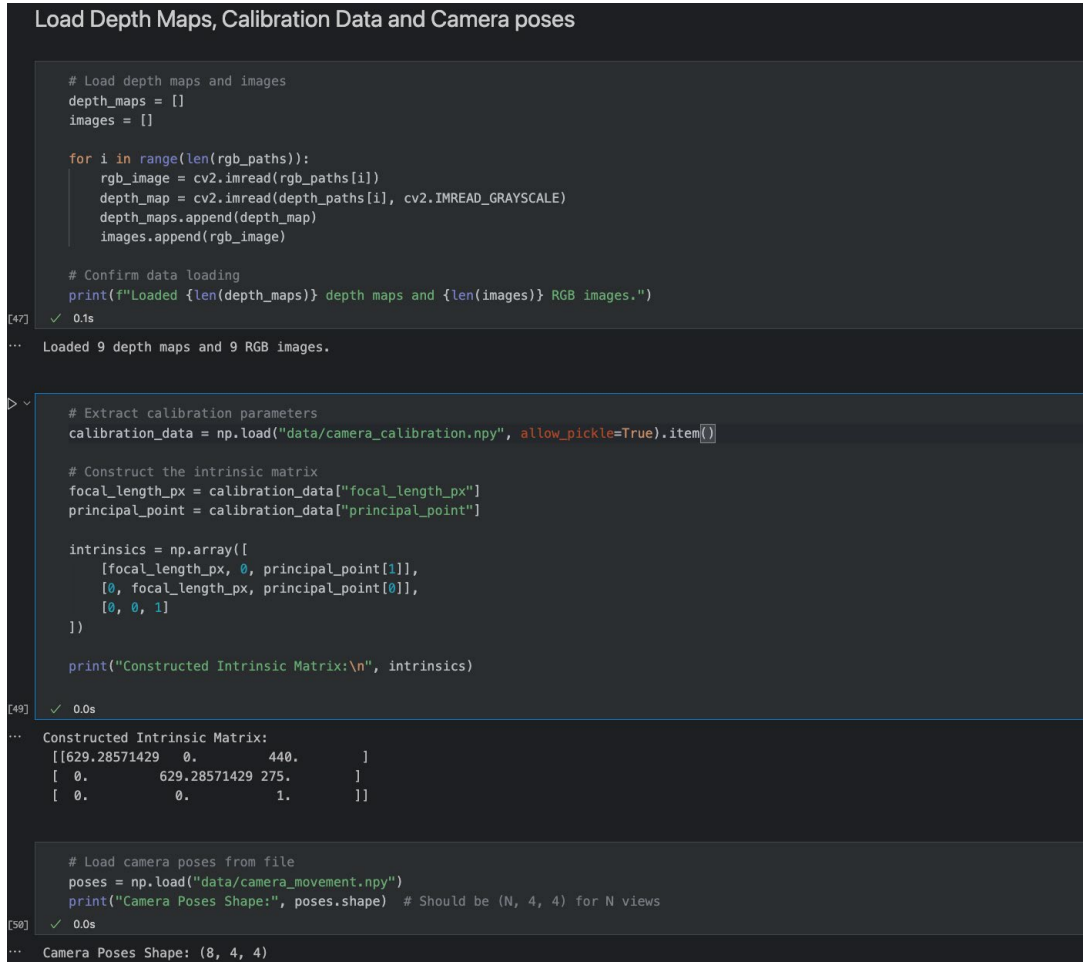
- Loaded 9 depth maps and RGB images.
- Converted images to grayscale for depth processing.

Camera Calibration:

- Loaded camera calibration parameters from file.
- Constructed intrinsic matrix using focal lengths and principal points.

Camera Poses:

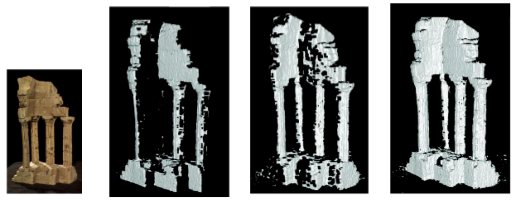
- Loaded 8 camera pose matrices to represent transformations between views.



Scene Representation – Depth Map

- **Properties**
  - Usually one depth map per view (by matching and triangulating features in small sets of images)
  - No sampling of 3D space
  - Simple
  - Suitable for multi-core
  - Depth maps have to be merged

(a) Real-image (b) Depth-map

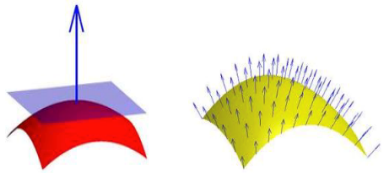


```python
def calculate_depths(feature_points, depth_map, reference_feature, reference_depth):
    """
    Calculate depths for 2D points based on the depth proportion from a reference feature.

    Args:
        feature_points (list of list): List of 2D points [u, v].
        depth_map (numpy.ndarray): Depth map (H, W).
        reference_feature (list): Reference feature [u, v].
        reference_depth (float): Z-depth of the reference feature.

    Returns:
        list: Calculated depths for the feature points.
    """
    depths = []
    ref_depth_value = depth_map[reference_feature[0], reference_feature[1]]

    for point in feature_points:
        if point == [-1, -1]:  # Ignore invalid points
            depths.append(None)
        else:
            depth_value = depth_map[point[0], point[1]]
            if depth_value == 255 or depth_value == 0:  # Ignore invalid values
                depths.append(None)
            else:
                depth = reference_depth * (depth_value / ref_depth_value)
                depths.append(depth)

    return depths
```

✓ 0.0s

Scene Representation – Point Cloud

- **Properties**
  - Local planar approximation of the surface at each point
  - Usually a normal vector is associated to each point
  - Sparse to dense reconstruction (region growing)
  - Suitable for multi-core
  - May be noisy



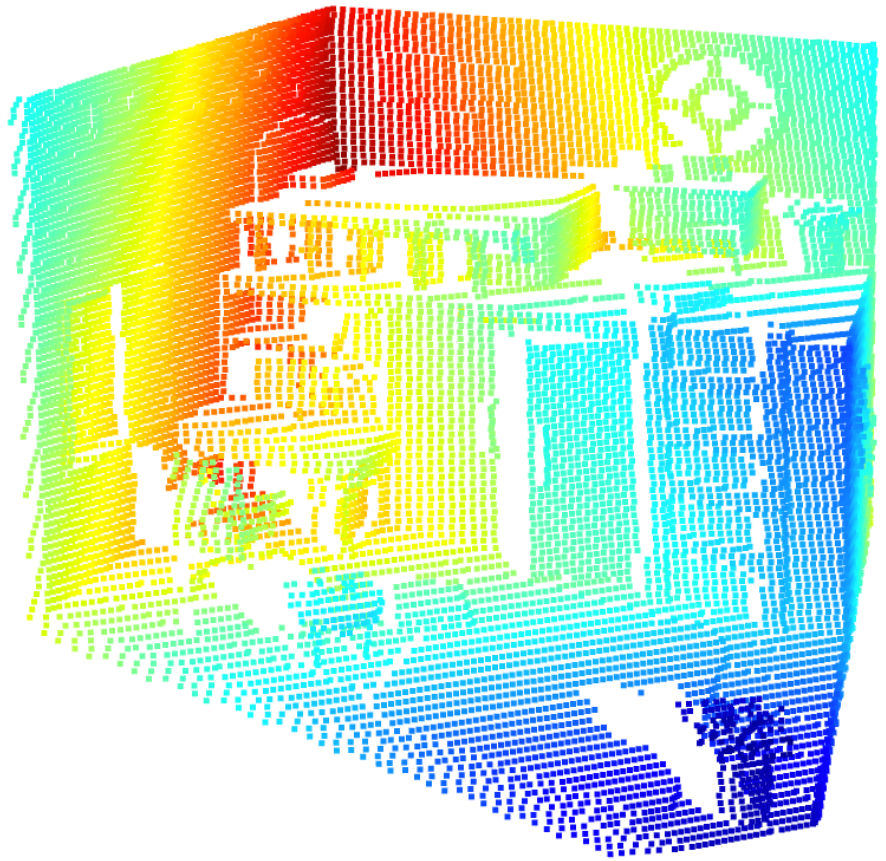
```python
def generate_3d_points(points_2d, depths, intrinsics):
    """
    Generate 3D points from 2D points and their depths.

    Args:
        points_2d (list of list): List of 2D points [u, v].
        depths (list): List of calculated depths for each point.
        intrinsics (numpy.ndarray): Camera intrinsic matrix (3x3).

    Returns:
        list: List of 3D points [x, y, z].
    """
    fx, fy, cx, cy = intrinsics[0, 0], intrinsics[1, 1], intrinsics[0, 2], intrinsics[1, 2]
    points_3d = []

    for point, depth in zip(points_2d, depths):
        if depth is None:
            continue
        u, v = point
        x = (u - cx) * depth / fx
        y = (v - cy) * depth / fy
        points_3d.append([x, y, depth])

    return points_3d
```

✓ 0.0s

Sparse Reconstruction

Pipeline Run to create Sparse Reconstruction

```python
# Generate 2D points grid
dataset_height = range(1, 551, 5)
dataset_width = range(1, 881, 5)
points_2d = [[h, w] for h in dataset_height for w in dataset_width]

# Reference feature and depth
reference_feature = feature_correspondence_matrix[0][0]  # First feature in the first view
reference_depth = points_3d[0][2]  # Z-depth of the first triangulated feature

# Calculate depths
depths = calculate_depths(points_2d, depth_maps[0], reference_feature, reference_depth)

# Generate 3D points
points_3d_first_view = generate_3d_points(points_2d, depths, intrinsics)

sparse_cloud = o3d.geometry.PointCloud()
sparse_cloud.points = o3d.utility.Vector3dVector(points_3d_first_view)

# Visualize the sparse point cloud
o3d.visualization.draw_geometries([sparse_cloud])
```



Dense Reconstruction- Partial 3D point clouds

Partial Point Clouds:

- Generated individual dense point clouds for each view using depth maps and RGB images.
- Incorporated camera intrinsics and poses for accurate 3D point projection.

Foundation for Dense Reconstruction:

- These partial point clouds represent segments of the scene from different viewpoints.
- Serve as building blocks to create a unified dense 3D model.

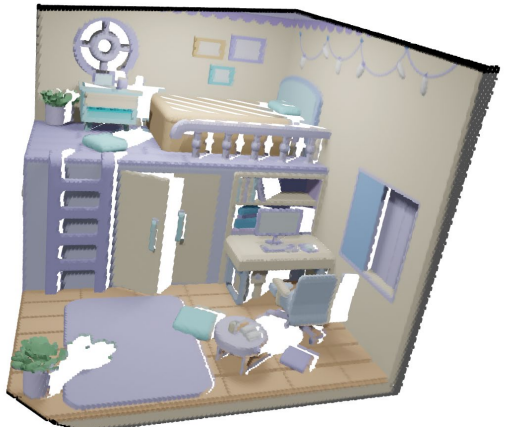
Next Step:

- Merge these partial point clouds using registration techniques like ICP.
- Achieve a complete, seamless dense reconstruction.

Dense Reconstruction- Partial 3D point clouds

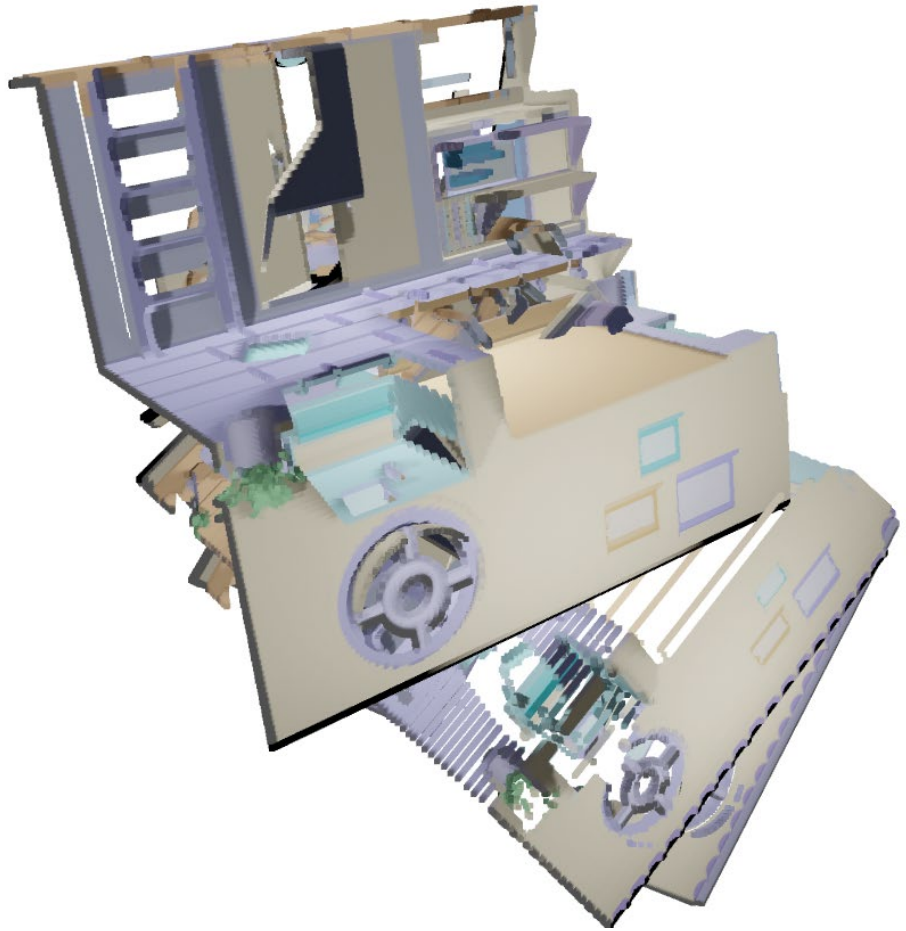
```python
def generate_dense_points(depth_map, rgb_image, intrinsics, pose, max_depth):
    """
    Generate a dense point cloud from a single depth map and RGB image.

    Args:
        depth_map (numpy.ndarray): Depth map (H, W).
        rgb_image (numpy.ndarray): RGB image (H, W, 3).
        intrinsics (numpy.ndarray): Camera intrinsic matrix (3x3).
        pose (numpy.ndarray): Camera pose matrix (4x4).
        max_depth (float): Maximum depth value for the current depth map.

    Returns:
        numpy.ndarray: 3D points (N, 3).
        numpy.ndarray: RGB colors (N, 3).
    """
    h, w = depth_map.shape
    fx, fy, cx, cy = intrinsics[0, 0], intrinsics[1, 1], intrinsics[0, 2], intrinsics[1, 2]

    points = []
    colors = []

    for v in range(h):
        for u in range(w):
            z = depth_map[v, u] / 255.0 * max_depth  # Scale depth to real-world units
            if z > 0 and z < max_depth:  # Ignore invalid or far points
                # Back-project to camera coordinates
                x = (u - cx) * z / fx
                y = (v - cy) * z / fy
                point_cam = np.array([x, y, z, 1])

                # Transform to world coordinates
                point_world = np.dot(pose, point_cam)
                points.append(point_world[:3])

                # Get color from RGB image
                colors.append(rgb_image[v, u, :])

    return np.array(points), np.array(colors)

# Generate partial dense point clouds
dense_points = []
dense_colors = []
for i in range(len(depth_maps)-1):
    max_depth = np.max(depth_maps[i]) / 255.0 * 10.0  # Calculate max depth per depth map
    points, colors = generate_dense_points(depth_maps[i], images[i], intrinsics, poses[i], max_depth)
    dense_points.append(points)
    dense_colors.append(colors)
    view_partial_point_clouds(dense_points, dense_colors)


# Combine all dense points into a single point cloud
final_dense_points = np.vstack(dense_points)
final_dense_colors = np.vstack(dense_colors)
```

✓ 11m 2.3s

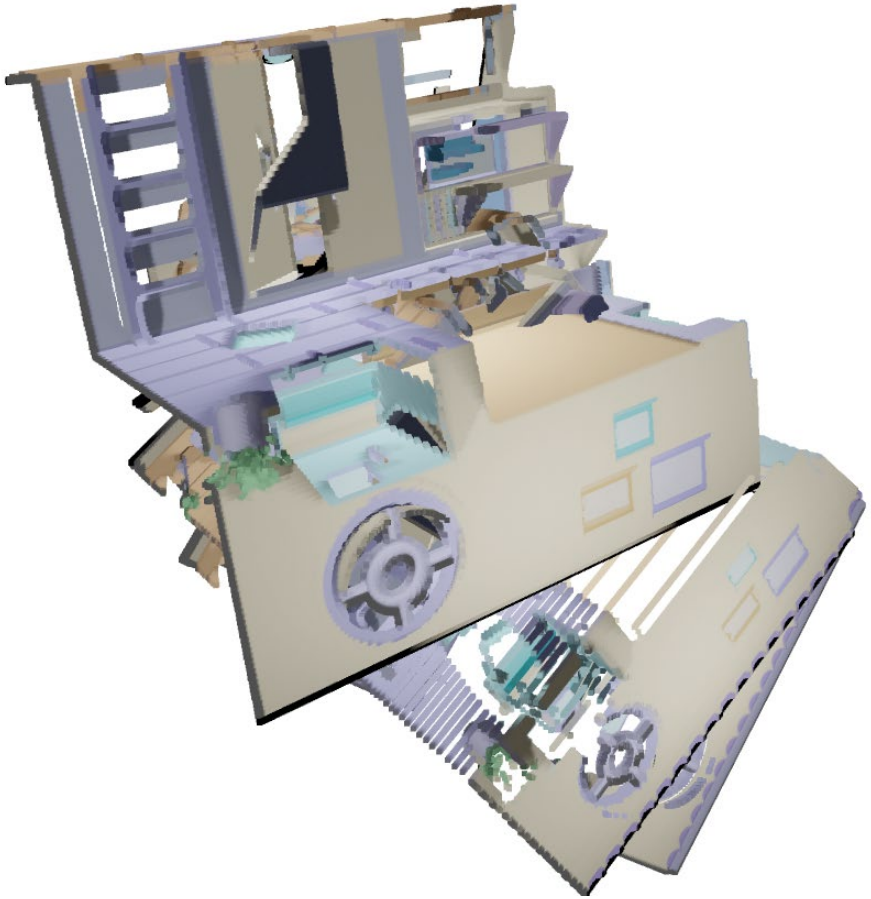


Dense Reconstruction

```python
import open3d as o3d

# Step 1: Normalize Depth Maps
def normalize_depth_maps(depth_maps):
    depth_maps_real = []
    for depth_map in depth_maps:
        max_depth = np.max(depth_map) / 255.0 * 5.0  # Adjust scaling factor
        depth_maps_real.append(depth_map / 255.0 * max_depth)
    return depth_maps_real

depth_maps_real = normalize_depth_maps(depth_maps)

# Step 2: Generate Dense Point Clouds for All Views
def generate_dense_points(depth_map, rgb_image, intrinsics, pose):
    h, w = depth_map.shape
    fx, fy, cx, cy = intrinsics[0, 0], intrinsics[1, 1], intrinsics[0, 2], intrinsics[1, 2]
    points = []
    colors = []
    for v in range(h):
        for u in range(w):
            z = depth_map[v, u]
            if z > 0 and z < 10.0:  # Ignore invalid or far points
                x = (u - cx) * z / fx
                y = (v - cy) * z / fy
                point_cam = np.array([x, y, z, 1])
                point_world = np.dot(pose, point_cam)
                points.append(point_world[:3])
                colors.append(rgb_image[v, u, :])
    return np.array(points), np.array(colors)

dense_points = []
dense_colors = []
for i in range(len(depth_maps)-1):
    points, colors = generate_dense_points(depth_maps_real[i], images[i], intrinsics, poses[i])
    dense_points.append(points)
    dense_colors.append(colors)

# Step 3: Merge and Align Partial Point Clouds
def merge_and_align_point_clouds(dense_points, dense_colors):
    merged_cloud = None
    for i, (points, colors) in enumerate(zip(dense_points, dense_colors)):
        cloud = o3d.geometry.PointCloud()
        cloud.points = o3d.utility.Vector3dVector(points)
        cloud.colors = o3d.utility.Vector3dVector(colors / 255.0)
        if merged_cloud is None:
            merged_cloud = cloud
        else:
            icp_result = o3d.pipelines.registration.registration_icp(
                cloud, merged_cloud, max_correspondence_distance=0.05,
                estimation_method=o3d.pipelines.registration.TransformationEstimationPointToPoint()
            )
            cloud.transform(icp_result.transformation)
            merged_cloud += cloud
    return merged_cloud

# Merge and refine alignment
final_dense_cloud = merge_and_align_point_clouds(dense_points, dense_colors)

# Step 4: Visualize and Save the Final Dense Reconstruction
o3d.visualization.draw_geometries([final_dense_cloud])
o3d.io.write_point_cloud("refined_dense_reconstruction.ply", final_dense_cloud)
print("Refined dense reconstruction saved as 'refined_dense_reconstruction.ply'.")
```



Final Result



Thank you for your time